

VICI—Virtual Machine Introspection for Cognitive Immunity

Timothy Fraser

The Microsoft Corporation tfraser@microsoft.com

Matthew R. Evenson

The Microsoft Corporation mevenson@microsoft.com

William A. Arbaugh

The Microsoft Corporation william.arbaugh@microsoft.com

Abstract

When systems are under constant attack, there is no time to restore those infected with malware to health manually—repair of infected systems must be fully automated and must occur within milliseconds. After detecting kernel-modifying rootkit infections using Virtual Machine Introspection, the VICI Agent applies a collection of novel repair techniques to automatically restore infected kernels to a healthy state. The VICI Agent operates without manual intervention and uses a form of automated reasoning borrowed from robotics to choose its best repair technique based on its assessment of the current situation, its memory of past engagements, and the potential cost of each technique. Its repairs have proven effective in tests against a collection of common kernel-modifying rootkit techniques. Virtualized systems monitored by the VICI Agent experience a decrease in application performance of roughly 5%.

1. Introduction

Today's Internet-facing systems are under constant attack. Upon gaining full administrative control of a compromised system, sophisticated adversaries install kernel-modifying rootkits, a form of malware that enables adversaries to hide their presence from a system's legitimate users and remain in clandestine control for days, weeks, or months [7, 14]. Kernel-modifying rootkits change the behavior of kernels by modifying their instructions and run-time state, forcing them to actively conceal the adversary's presence. Traditional user-mode intrusion detection programs that depend on correct kernel behavior cannot cope with kernel-modifying rootkits. New methods must be found to deal

with them.

Efforts such as XenKIMONO [22], VMwatcher [16], Lycosid [17], and others have explored the use of Virtual Machine Introspection [11] to detect kernel-modifying rootkits and have shown promising results. However, detection is only half the battle—we must also develop repair techniques capable of disabling kernel-modifying rootkits and restoring infected kernels to health. XenKIMONO can call for manual intervention upon detecting a kernel-modifying rootkit, but in an environment where systems are under constant attack only fully-automated repair techniques capable of restoring kernels to health within milliseconds of infection can hope to keep mission-critical systems running for any useful length of time.

We present the Virtual-machine Introspection for Cognitive Immunity (VICI) Agent prototype, the first introspection-based system to follow rootkit detection with completely automated repair functionality designed to restore infected kernels to health within milliseconds. The VICI Agent periodically examines the state of a running kernel using a collection of diagnostic functions designed to detect the kinds of modifications rootkits typically make to change kernel behavior. Upon detecting such a modification, the VICI Agent chooses one of several repair actions to undo it, disable the rootkit, and restore proper kernel behavior.

Different kernel-modifying rootkits may require different repair actions. Some kernel-modifying rootkits may actively attempt to overcome repairs, and may require extreme measures to eradicate. Extreme measures may come with some cost in terms of lost useful state and availability. In order to enable infected systems to keep performing their mission to the greatest extent possible, autonomous repair agents like the VICI

Agent must reason about tradeoffs between repair effectiveness and cost.

The VICI Agent has a collection of repair actions, some designed to deal with simple rootkits, others geared towards tackling more complex examples. Upon detecting an infection, the VICI Agent must autonomously choose the cheapest relevant repair action that seems likely to defeat the rootkit based on its memory of recent engagements. We have adapted the Subsumption architecture from robotics [4, 5, 6] to provide the VICI Agent with the automated reasoning capability it needs to make these choices.

In order to control cost, the VICI Agent begins by choosing its least expensive relevant repair action. If the infection persists, the VICI Agent *escalates* through a series of increasingly expensive repair actions until it either finds one that removes the infection or runs out of choices. Once found, the VICI Agent remembers which repair action did the trick for an interval of time. If a rootkit re-infects the kernel during that interval, the VICI Agent saves time by applying the previously-effective repair action immediately rather than escalating to find it again. A complimentary *de-escalation* behavior allows VICI to gradually return to cheaper repair actions if infections decrease in complexity or cease.

In our experiments, the VICI Agent’s simplest repair actions have proven sufficient to undo modifications of the kind made by the well-known Adore-ng, Override, and SuKIT rootkits. When configured to run its diagnostics once every 50 milliseconds, our benchmarks indicate that the VICI Agent reduces overall application performance on the monitored virtual machine by slightly more than 5% in comparison to an identical virtual machine without the VICI Agent.

The remainder of this paper is organized as follows: Section 2 explains how rootkits modify kernels in order to change their behavior. Section 3 describes the architecture of the VICI Agent and explains its basic operation. It is followed by three sections that focus in greater detail on what the VICI Agent’s diagnostics and repair actions do (Section 4), how it uses automated reasoning to decide which repair to use when (Section 5), and how it uses the subsumption architecture to implement its automated reasoning (Section 6). Section 7 discusses related work. Section 8 presents benchmark results measuring how much virtualization and the operation of the VICI Agent reduces the performance of applications on the virtualized system. In section 9 we discuss the limitations of our approach and how they might be addressed by future work, and in section 10 we present our conclusions.

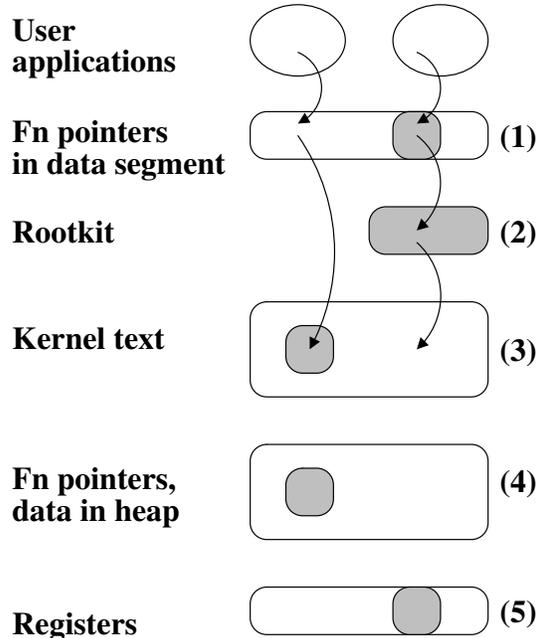


Figure 1. Some ways rootkits modify kernels.

2. Kernel-modifying rootkits

Kernel-modifying rootkits modify the behavior of the infected system’s kernel, causing it to lie to the system’s authorized users and administrators. Whenever an authorized user asks the kernel to produce a list of presently-existing objects such as files, processes, sockets, or dynamically loaded kernel modules, the kernel-modifying rootkit will carefully remove those objects belonging to the adversary from the lists, thereby hiding the adversary’s presence. Some rootkits also provide secondary functionality to the adversary, such as backdoors or keyboard sniffers.

The diagram in Figure 1 shows some of the ways in which kernel-modifying rootkits change the kernel’s behavior. The arrows in the diagram approximate the control flow of two user processes (ovals) asking the kernel for service through system calls. White rectangles represent aspects of kernel state, shading indicates modifications made by rootkits.

In a healthy kernel, the system call interrupt handler looks up the address of the kernel function intended to handle a given call in an array of function pointers called the system call vector, and jumps there. Some rootkits replace the addresses of kernel functions in this vector (1) with the addresses of their own functions (2), diverting kernel control flow through their own malicious code that filters the results returned to the user process. Other rootkits directly modify the kernel’s be-

havior by tampering with its text (executable instructions, 3). There are many function pointers for rootkits to modify, some in the kernel’s data segments and some in the heap (4). Rootkits may also modify register values (5) or kernel or process virtual address translation tables to achieve their ends.

We focus on kernel-modifying rootkits because they are hard to detect with traditional sensors implemented as user programs that depend on the kernel telling the truth. Even sensors implemented as kernel-mode drivers face an arms race against rootkits designed to seek them out and disable them. Sensors must be isolated from the kernel they defend, perhaps by virtualization as the VICI Agent is.

3. Architecture and assumptions

The diagram in Figure 2 shows the architecture of the VICI Agent and its place in a deployed system. On the right side of the diagram is a virtual machine supported by the Xen hypervisor using its Hardware Virtual Machine mode, not paravirtualization [10]. This virtual machine runs an unmodified Commercial Off-The-Shelf GNU/Linux operating system and supports the overall system’s mission applications. It deals with clients over the network, making it vulnerable to rootkit infection.

On the left side of the diagram is Xen’s “Domain 0” administrative operating system that provides administrators with control of the “real” machine and all of its virtual machines. The VICI Agent runs as a privileged user process on this operating system. It uses hypercalls provided by the Xen hypervisor to pause the virtual machine for examination and later unpause it, read and write the virtual machine’s registers, and memory-map the virtual machine’s “physical” memory into the VICI Agent’s address space where the VICI Agent can then read and write it. Most of this functionality was provided by Xen; a small patch added the rest.

Xen is designed to isolate Domain 0 from the virtual machine. To the extent that this isolation is effective and to the extent that the VICI Agent is careful in its interactions with the virtual machine, the VICI Agent can be expected to continue to operate correctly even if the virtual machine’s kernel is thoroughly compromised by a rootkit.

The VICI Agent runs in a loop, making a single “scan” on each iteration. At the beginning of each scan, the VICI Agent runs its diagnostics to examine the virtual machine’s kernel (1). These diagnostics are designed to detect many of the kinds of tampering shown in Figure 1, and enable the VICI Agent to detect the presence of kernel-modifying rootkits. At the end of

each scan, if the diagnostics found any problems, the VICI Agent attempts to modify the state of the virtual machine and/or its kernel in a way that will restore the kernel to health (3). The arrows in the diagram emphasize that the VICI Agent’s code runs in Domain 0, outside of the virtual machine, and gains access to the virtual machine’s memory and register state using hypercalls provided by Xen. This access is one-way; it is not available to processes on the virtual machine. The VICI Agent neither runs code nor stores state on the virtual machine.

In the middle of each scan, the VICI Agent considers both the results of the present scan’s diagnostics and those of the previous scan (2). If the VICI Agent sees that problems were found in both scans, it decides the previous scan’s repairs failed, and learns from this failure by escalating: that is, by biasing its future choices towards more expensive repair actions.

The VICI Agent is concerned only with kernel-modifying rootkits. However, it could be used in conjunction with user-mode malware detection programs. This arrangement could be mutually beneficial: the VICI Agent could give the user-mode programs greater assurance that their kernel is not lying to them, and the user-mode programs could examine high-level filesystem state and network traffic that is difficult to decipher from VICI’s relatively low level of abstraction.

The VICI Agent is a research prototype. In order to direct more of our development resources towards prototyping novel features, we made several labor-saving assumptions: First we did not harden Xen or the Domain 0 operating system. In a real deployment, one might choose to use a custom special-purpose hypervisor and Domain 0 operating system rather than the general-purpose Xen Hypervisor. By implementing only those features specifically needed by the VICI Agent, it would be easier to harden.

Second, we assume that the virtual machine’s kernel remains uninfected long enough for the VICI Agent to take a snapshot of it in its healthy state shortly after boot. In a real deployment, one might prefer to examine binaries off of known-good read-only installation media and emulate the loading process in order to determine what a healthy kernel looks like, instead. This would be an effective defense against rootkits that use modified Master Boot Records to install themselves before any snapshot could be taken.

Note that the VICI Agent doesn’t care how the rootkit is installed. We assume the adversaries have full administrative control of the virtual machine’s operating system and can install as they please. The VICI Agent’s diagnostics look for the modifications rootkits make to kernel state after installation.

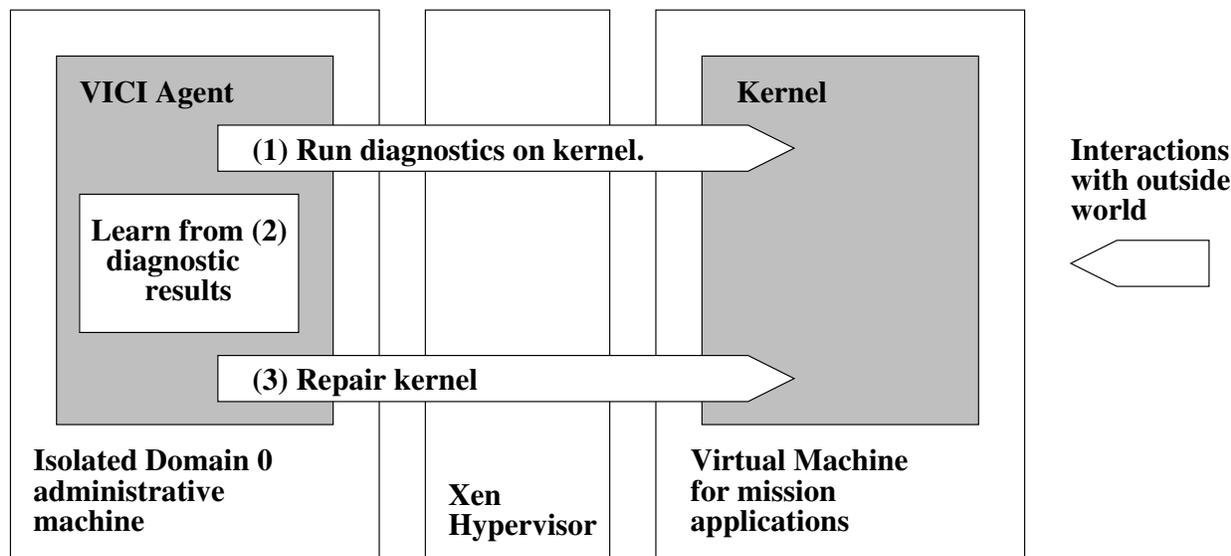


Figure 2. The VICI Agent protects a kernel running in a virtual machine.

4. Diagnosis and repair

The VICI Agent uses a form of Virtual Machine Introspection [11] to run its diagnostics on the virtual machine’s kernel. To detect tampering with the kernel’s instructions, it computes an MD5 checksum [24] of the kernel’s text and compares the result against the expected value for a healthy kernel. It performs similar checksum comparisons for pages of Loadable Kernel Module text. To detect changes to important function pointers, registers, and constants, the VICI Agent compares the values observed in kernel memory with the values expected for a healthy kernel.

The VICI Agent’s diagnostics cover some of the aspects of kernel state listed in figure 1, specifically kernel and module text, over 9000 kernel and module function pointers, and a few commonly-targeted registers. There are also diagnostics designed to detect the introduction of unapproved packet handlers and a tampering attack on the Linux kernel’s pseudo-random number generator described by Baliga and others [3]. We do not claim this coverage is complete, as explained in section 9.

The VICI Agent has a fixed collection of repair actions. The simplest repair action is “Surgical” repair, so-called because it is capable of repairing the most common forms of rootkit tampering without harming the repaired kernel. The rest of the repair actions represent stronger medicine with correspondingly greater costs. The VICI Agent tries to hold these more costly repair actions in reserve until faced with a rootkit that the less-costly repair actions cannot handle. Each repair action is described below:

The Surgical repair action simply writes correct values to text locations, variables, and registers that the diagnostics have found incorrect. In practice, this simple low-cost strategy has proven sufficient to defeat the malicious techniques used by the rootkits we have collected from the Internet. However, we have developed a series of more complex test rootkits that require more extreme repairs.

The Core War repair action is designed to handle more complex rootkits that borrow the VICI Agent’s own techniques to defeat Surgical repair. One of our test rootkits uses a kernel thread to monitor the improper values it writes to the kernel’s system call vector. Whenever the VICI Agent’s Surgical repair restores the proper values, the kernel thread writes the improper values back again, negating the repair.

Upon witnessing the failure of its Surgical repair, the VICI Agent moves to the Core War repair action inspired by the classic game of Core War [9]. The Core War repair finds the rootkit’s text by following the improper pointer from the system call vector, and then “neuters” the rootkit by re-writing its code to jump immediately to the kernel’s proper function without performing any of the rootkit’s malicious functionality. Although control still flows through the rootkit, its malicious functionality is removed and the VICI Agent’s purpose is served. Although our code-rewriting seeks to avoid results that would accidentally crash the kernel, we reserve Core War for cases in which the simpler and safer Surgical repair action fails.

The Hitman repair action is designed to handle even more sophisticated rootkits that are capable of defeating Core War. Another of our test rootkits uses a kernel thread to keep not only its modifications to the kernel's system call vector in place, but also to re-write its own instructions if it finds them neutered.

The VICI Agent turns to Hitman when it witnesses the failure of Core War. Hitman guesses which threads may be aiding the rootkit and kills them. Its method of determining which threads to kill is approximate and often includes a few innocent threads along with the guilty. Like Core War, the Hitman repair action uses the improper function pointer values in the kernel's system call vector to locate the rootkit's malicious functions in kernel memory. Hitman calculates the start and end address of the kernel page or pages that hold those functions. It then examines the top 64 words from each thread's kernel stack. If it finds a word-sized value that falls within the address ranges of those pages, it kills that thread.

This approach often kills kernel threads that are aiding rootkits because they store saved instruction pointers within the target address range near the tops of their kernel stacks when they give up the CPU. However, this approach also kills any other threads that happen to have a matching word-sized pattern of bits on their stack, whether it is a saved instruction pointer or not. Although this imprecision is costly, it is still worth attempting the Hitman repair since it may save us from having to move on to the next two repairs which are even more extreme and will surely cause greater loss of useful state.

The Checkpoint repair action is different from Core War and Hitman in that it does not attempt to counter a specific rootkit strategy. Instead, if prior repairs fail for whatever reason, the Checkpoint repair action simply restores the virtual machine to a previously-checkpointed state. In a deployment, we imagine administrators might checkpoint the virtual machine periodically. This repair action hopes to restore the kernel to health by returning it to a previously checkpointed state that predates the time of infection.

The Reboot repair action is the VICI Agent's option of last resort. As a deployed system is apt to have storage capacity for only a finite number of checkpoints, it is possible that repeated Checkpoint restores may eventually exhaust this supply. In this situation, further failures of cheaper repairs will cause the VICI Agent to reboot the virtual machine, returning it to a very early state that is hopefully free of infection. In our prototype, reboot is implemented by restoring a checkpoint

of the virtual machine meant to be taken just after operating system install-time. It loses all useful application state. If for some reason Reboot cannot restore the kernel to health, the VICI Agent resorts to continuously rebooting the machine. Neither the rootkit nor the system's mission applications make progress, but the VICI Agent has succeeded in making the presence of the rootkit known.

The Core War and Hitman repair actions are presently implemented only for attacks on the system call vector. Difficult attacks on other aspects of the kernel's state cause the VICI Agent to skip directly to its Checkpoint repair action.

In a real deployment, administrators might prefer to avoid the more expensive repair actions like Checkpoint and Reboot, preferring to shutdown or call for manual intervention instead. On the other hand, these repairs might be appropriate for nodes in systems like Chord [28] or MapReduce [8] that have automatic protocols in place to tolerate or recover from node failures.

5. Control and learning

The VICI Agent is designed to apply its cheapest repairs first, escalating to the more expensive repair actions only when the cheaper ones have failed. The diagram in Figure 3A represents this escalation behavior. The diagram represents a series of VICI Agent scans as a series of rectangles. Shaded rectangles represent scans where the VICI Agent's diagnostics indicate problems. The height of the rectangles increase with the costliness of the VICI Agent's repair action choices.

In scan 2, we can imagine the VICI Agent detected tampering in the kernel's system call vector, and applied its Surgical repair action. But in scan 3, the problem remains and the VICI Agent escalates to its Core War repair action. The problem still persists in scan 4, so the VICI Agent escalates further to Hitman. Hitman finally repairs the kernel, and scan 5 reveals a healthy kernel. By reserving the expensive Hitman repair action until last, and by not moving on to the even more expensive Checkpoint and Reboot repair actions once Hitman proved effective, the VICI Agent meets its goal of avoiding repair cost when possible.

We say that the VICI Agent has an "anger level". Once it escalates to Hitman in scan 4 of the previous example, the VICI Agent remains at that "anger level" for a constant number of scans (10 in this example). If the infection recurs during that interval, rather than performing the same escalation through Surgical and Core War to Hitman as before, the VICI Agent instead applies Hitman immediately, as in scan 9. In situations where the kernel is being repeatedly infected by

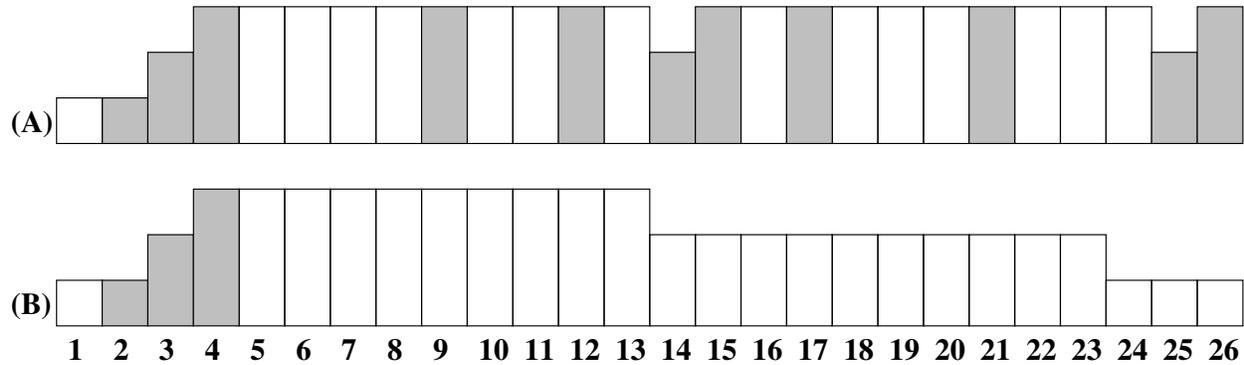


Figure 3. The VICI Agent escalates its “anger level” in response to a difficult rootkit infection. Shaded bars represent scans in which the Agent detects tampering. In (A) the Agent remains escalated in response to repeated re-infections. In (B) the infections cease and it eventually de-escalates.

a rootkit of sufficient sophistication to require Hitman repair, staying at an elevated anger level optimizes the VICI Agent’s response for time by skipping the presumably ineffective cheaper repairs.

When 10 scans have elapsed since the initial escalation, the VICI Agent will de-escalate—that is, decrease its anger level by one. If the infection recurs at this point as in scans 14 and 15, the VICI Agent escalates back to the higher level. If the infection does not recur, the VICI Agent de-escalates back to its lowest anger level in a series of 10 scan steps, as shown in Figure 3B. The combination of escalation and de-escalation enables the VICI Agent to provide a limited form of Cognitive Immunity [23]—the ability to adjust its defenses in response to changing attacks.

6. Control implementation

Automatic detection–repair systems like the VICI Agent are examples of continuous or reactive planning agents that spend their entire runtimes looping again and again through phases of sensing, planning, and execution. Continuous planning agents such as IPEM [1] have used partial-order planning to map sensor reports to appropriate actions. Reactive planning agents like PRS [12] have used Hierarchical Task Networks. However, both of these strategies require the designer to create internal abstract representations of the problem domain on which the planners will operate—a potentially difficult and error-prone task. This requirement is not suitable for dealing with kernel-modifying rootkits, as one mistake or bad assumption about rootkit behavior or kernel state in this abstract representation will give rootkits an avenue by which they can defeat the agent. Furthermore, future advances in malicious rootkit behavior are difficult to predict.

The “Subsumption Architecture” is an alternative approach originally intended for the development of autonomous mobile robots that exhibit insect-like behavior [4, 5, 6]. Unlike the other approaches, it requires no potentially troublesome abstract representation. Instead, following the maxim that “the world is its own best representation,” the subsumption approach has the agent’s loop continuously poll sensors to determine the state of the world and react to their reports directly, rather than indirectly through a model of the world that may or may not be complete enough for the task. Subsumption also permits the agent to maintain state across loop iterations, enabling it to remember and learn from past experiences. Subsumption-based agents have been concisely described as “a particular kind of reflex agent with state” [25].

In addition to the above advantages, the subsumption architecture also permits decision-making rapid enough to meet our goal of achieving repair within milliseconds of diagnosis, and is amenable to incremental development and easy extension as new kinds of kernel-modifying rootkit threats come to light.

The subsumption architecture advocates implementing and testing control schemes for mobile robots incrementally in a sequence of layers: first a layer to avoid collisions, then a second to build upon and sometimes override the first to make the robot wander while avoiding collisions, and finally a third layer to shape the lower two’s behavior into purposeful movement toward some distant destination.

The VICI Agent adapts this architecture by implementing each of its repair actions as a layer, with higher layers observing the effectiveness of the lower ones and applying their more expensive repairs only when the lower layers’ cheaper ones have failed.

7. Related Work

The VICI Agent borrows techniques from a number of systems that use virtual machine introspection to monitor the state of virtualized systems [11, 2, 17, 16, 22]. To their detection functionality the VICI Agents adds fully-automated kernel repair techniques and the automated reasoning capability needed to apply these repairs in a way that avoids losing useful system state and availability when possible.

The phrase “virtual machine introspection” was coined by Garfinkel and Rosenblum to describe the operation of their Livewire prototype which used a hypervisor to implement a host-based intrusion detection system for virtual machines [11]. The hypervisor allowed them to locate their monitor outside of the virtual machine, thereby protecting it from tampering. Livewire detected intrusions both in the kernel and in user-mode applications. Although it could always provide some value, it could not operate at full effectiveness if an adversary managed to modify the monitored system’s kernel state to make Livewire’s operation difficult. The VICI Agent adds repair and automated reasoning features to Livewire-like diagnostic functionality.

Building on the ideas of Livewire, several other efforts have used hypervisors to monitor systems running in virtual machines for misbehavior. Asrigo and others have used a variety of virtualization approaches, including the Xen hypervisor used by the VICI Agent, to instrument the kernels of virtualized honeypot machines so that they produce better records of user-mode process activity [2]. The Asrigo approach focuses on user-mode activity rather than on the integrity of kernels, as the VICI Agent does.

The Lycosid system [17], however, uses a hypervisor-based monitor to detect processes hidden by kernel-modifying rootkits—a goal closely related to the VICI Agent’s. Lycosid uses “cross-view” validation to detect hidden processes [29]. It gathers process-related data from different sources within the system and compares them. If a rootkit has tampered with some sources but not others, this comparison will reveal inconsistencies that indicate the presence of a hidden process. The present VICI Agent prototype’s diagnostics focus mainly on detecting control-flow and text changes that alter the kernel’s behavior. Although these existing diagnostics might discover the mechanism a rootkit puts in place to hide processes, it seems likely that the VICI Agent’s effectiveness could be improved by adding a Lycosid-like diagnostic for detecting the hidden processes themselves.

The VMwatcher system [16] uses a hypervisor to gather information about a system running in a

virtual machine. This information is necessarily in terms of low-level kernel abstractions (pages, registers, disk blocks). VMwatcher introduces a technique called Guest View Casting to recast this low-level information into equivalent higher-level abstractions (processes, files) that can be fed to traditional user-mode host-based intrusion detection systems. VMwatcher uses this technique to isolate a variety of commercial user-mode malware detection programs from the virtualized system they monitor, thereby protecting them from tampering. In addition, it also performs cross-view validation like Lycosid.

The XenKIMONO system [22] combines rootkit detection functionality similar to the VICI Agent’s basic diagnostics with cross-view validation similar to Lycosid and VMwatcher. Unlike the VICI Agent, XenKIMONO also monitors the integrity of some critical user-mode daemons. Both XenKIMONO and the VICI Agent recognize the need to take some kind of ameliorative action upon detecting a rootkit infection. Administrators can configure XenKIMONO to stop or pause an infected virtual machine and call for manual intervention, or checkpoint its state for later forensic analysis. Rather than wait for manual intervention, the VICI Agent tries to repair infected kernels itself, automatically and within milliseconds.

Some hypervisor-based efforts have explored intrusion prevention rather than detection and repair [18, 30, 26]. While these systems have demonstrated effective prevention, those that report benchmark results show overheads far greater than those imposed by the VICI Agent.

The Manitou system [18] inserts special page-fault handling code into the hypervisor that examines each page of executable text before it is first paged in for execution. Manitou computes a hash of each page and compares it to a list that contains hashes for all authorized executable text pages. If the hash is not in the list, Manitou prevents execution of the page.

The UCON system [30] uses the Bochs IA-32 emulator to demonstrate an intrusion prevention technique that could be applied to hypervisors. The emulator performs an access control checks whenever it executes a machine instruction to write memory. If the write would violate the machine’s security policy, UCON prohibits the write. Configured with a security policy that prohibits writes to many of the same kernel data structures examined by the VICI Agent’s basic diagnostics, UCON has prevented the installation of many common rootkits in laboratory tests.

The SecVisor system [26] uses the Secure Virtual Machine feature of AMD CPUs to virtualize a machine’s MMU and IOMMU and pass control of their

operations to a very small hypervisor. The hypervisor enforces a page protection policy that is more strict than the one enforced by the traditional MMU. It prevents the writing of kernel text and the execution of kernel data by the CPU even when in kernel-mode or by devices doing DMA, thereby defeating rootkits that modify kernel instructions or depend on the execution of malicious code in the kernel’s heap. However, it does not prevent rootkits from changing kernel behavior by modifying data or CPU registers.

Although Manitou and UCON clearly provide useful preventive functionality, there are no published measurements of their performance overheads and it is consequently difficult to compare the costs of prevention to detection and repair. SecVisor’s authors report the overhead of Xen virtualization plus SecVisor on a kernel build benchmark similar to the one described in section 8 as being roughly 119%. Our equivalent measurement for Xen plus VICI is only roughly 42%.

Grizzard [13] and Petroni and Hicks [21] have explored hypervisor-based methods for countering attempts to change the behavior of kernels by modifying their control flow. Grizzard modifies the monitored kernel’s code to cause it to ask the hypervisor to perform a check whenever the kernel is about to jump or call to an address stored in a function pointer. The hypervisor checks to see whether or not the jump or call follows a branch in the kernel’s proper control-flow graph. If a rootkit has modified the function pointer to cause a jump not found in this graph, the hypervisor will detect the diversion. Petroni and Hicks explored a technique based on static analysis to achieve similar ends. We have incorporated part of their technique into the VICI Agent; it is described in section 9.

The above efforts use hypervisors to isolate their monitors from the virtualized systems they monitor. However, this virtualization reduces the systems’ performance. Several projects have explored the possibility of using special hardware to provide isolation without virtualization [31] by placing the monitor on a PCI card [19] or on one CPU of a dual-CPU machine [15]. Although they avoid virtualization overhead, these solutions cannot examine the registers of the CPU that runs the monitored system. Their inability to monitor useful rootkit targets such as CPU pointers to page tables makes the monitors vulnerable to a “dummy kernel” attack described in [27].

8. Performance

Table 1 compares the average time required to build the Linux 2.6.24 kernel in its default configuration under three conditions. The “No Xen” results refer to

	No Xen	Xen	Xen + VICI
average duration (s)	1782.17	2412.05	2537.35
standard deviation	4.64	3.34	3.89
penalty vs. no Xen		35.34%	42.37%
penalty vs. Xen			5.19%

Table 1. Kernel compile durations compared.

builds on a Lenovo T60 laptop with an Intel T7200 Core Duo 2GHz CPU and 2GB RAM running Debian GNU/Linux 4.0. The “Xen” results refer to building the kernel in a Xen 3.1 virtual machine running Debian GNU/Linux with 128MB of virtual RAM hosted on the same T60 laptop. The “Xen + VICI” results refer to builds on the same virtual machine with the VICI Agent running one scan every 50ms, no attacks, no repairs. The averages are computed over five trials each.

The table addresses both the overhead associated with virtualization and the additional overhead introduced by the VICI Agent. The performance of the VICI Agent’s diagnose-and-repair approach appears to compare favorably with the performance of SecVisor’s prevention-oriented alternative. Although our the cost of virtualization makes the VICI Agent’s overhead greater than that reported for hardware-based monitors like Copilot, our hypervisor-based approach permits us to examine CPU register state, thereby giving us the potential to avoid the most severe shortcoming of hardware-based schemes.

9. Limitations and future work

Although they have performed well in the laboratory, the diagnostics implemented in the present VICI Agent prototype do not examine all of the aspects of the virtual machine and kernel state that a rootkit might modify to its own advantage. The VICI Agent misses some aspects because the best diagnostic methods published to date require more than 250 milliseconds to execute (a time bound mandated by the project’s sponsor). In other cases, efficient diagnostic methods exist, but we have not yet implemented them. In particular, we have not yet implemented diagnostics to cover kernel and process page tables in a way that would detect the “dummy kernel” attack mentioned in section 7. Future work could ignore the 250 millisecond bound and incorporate more diagnostic methods to increase coverage.

In order to improve our diagnostic coverage of kernel function pointers, we have used part of Petroni and Hicks' State-Based Control Flow Integrity static analysis technique [21] to produce a list of kernel function pointers the VICI Agent's diagnostics should examine. This technique performs a static analysis of the Linux kernel source and outputs both a list of the function pointers that reside in the kernel's data segments, but also outputs C code that, when executed during a diagnostic, can find additional function pointers dynamically allocated in the kernel's heap.

Unfortunately, using hardware similar to our own, Petroni and Hicks reported that an average run of a prototype diagnostic function based on their technique took 1.78 seconds to examine all of the function pointers in the kernel's data segments and heap—far more than the VICI Agent's 250 millisecond requirement will permit. Consequently, we presently make use of only part of their technique: the list of function pointers in the kernel's data segments, and do not examine the function pointers in the heap.

Healthy kernels change the values of some of the function pointers in this list during runtime. To prevent our diagnostics from falsely reporting these changes as rootkit tampering, we have attempted to identify the changing function pointers by exercising the kernel and removing them from the list. Our list of kernel function pointers presently contains over 9000 entries.

In addition to its imperfect coverage, the VICI Agent is also hampered by the fact that it operates asynchronously with the virtual machine's kernel, and scans periodically only once every so many milliseconds. It is not hard to imagine a piece of malware that avoids detection by exploiting one or both of these problems: perhaps by making and immediately reversing a change between scans, or by changing a function pointer outside of the VICI Agent's present coverage. However, the VICI Agent is not intended to be a general malware detector. It is designed to detect and repair the effects of kernel-modifying rootkits—a very specific kind of malware whose purpose demands a set of peculiar attributes.

Momentary kernel state modifications are not apt to be of use to kernel-modifying rootkits, except perhaps when combined with a small number of strategically-placed long-term modifications. Adversaries intend rootkits to hide their presence over days, weeks, or months, and consequently rootkits must make at least one persistent modification to the kernel in order to gain control whenever they need it. Furthermore, adversaries depend on rootkits to hide their presence from all of the system's legitimate users. Peripheral, non-strategically-placed changes are insufficient for this task. In order to

hide the adversary's files, processes, and other resources from all other users at all times, we argue rootkits must modify at least one of several strategic points along the kernel's system call control flow path. Although the VICI Agent's diagnostic coverage is limited, we have sought to cover all of these strategic points.

There are a number of additional diagnostic techniques that could improve the effectiveness of the VICI Agent prototype, particularly in the area of data structures whose values naturally change during runtime and whose health consequently cannot be determined by a simple comparison against a single known-good value. Petroni and others' notion of semantic integrity checks may be applicable here, in particular their probabilistic technique for finding processes that have been hidden by strategic de-linking from kernel lists [20]. There is also room for improvement in the present prototype's repair actions. The Checkpoint and Reboot repair actions, for example, cannot complete within the VICI Agent's 250 millisecond time bound.

Throughout its development, we have greatly improved the VICI Agent capabilities by adapting and incorporating the techniques of others from a variety of fields, and there appears to be considerable room for future work in continuing to do so.

10. Conclusions

Systems that are under constant attack cannot wait for manual intervention to remedy malware infections. The VICI Agent uses a collection of novel repair actions to repair kernels modified by rootkits within milliseconds of diagnosis.

The VICI Agent's escalation and de-escalation behaviors enable it to reduce the negative impact of its repairs on the system by saving its most expensive repairs for extreme cases and relying on cheaper repairs when they will do. In our experiments, the present VICI Agent prototype's simplest repair actions have proven sufficient to undo modifications of the kind made by the well-known Adore-ng, Override, and SuKIT rootkits.

When configured to run its diagnostics once every 50 milliseconds, the VICI Agent reduces overall application performance on the monitored system by slightly more than 5% in comparison to an identical virtualized system without the VICI Agent. This overhead is less than half that of the only published measurements for prevention-oriented alternatives. The VICI Agent demonstrates the kind of rapid fully-automatic kernel repair necessary to keep Internet-facing systems running and free of kernel-modifying-rootkits despite being under continuous attack.

The authors would like to thank Peter Ferrie,

Michael Hicks, and Nick Petroni for their feedback on earlier drafts of this paper. This work was supported by DARPA/AFRL contract number FA8750-07-C-0008. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

References

- [1] J. A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proc. of the Seventh National Conference on Artificial Intelligence*, 1988.
- [2] K. Asrigo, L. Litty, and D. Lie. Using vmm-based sensors to monitor honeypots. In *2nd International Conference on Virtual Execution Environments*, 2006.
- [3] A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data (Short Paper). In *Proc. of the 2007 IEEE Symposium on Security and Privacy*, 2007.
- [4] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, (2), 1986.
- [5] R. A. Brooks. Engineering approach to building complete, intelligent beings. *Proc. of the SPIE—the International Society for Optical Engineering*, (1002), 1989.
- [6] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3), 1991.
- [7] D. Brumley. invisible intruders: rootkits in practice. ;login: *The Magazine of USENIX and SAGE*, Sept. 1999.
- [8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing in Large Clusters. In *6th USENIX Symposium on Operating System Design and Implementation*, 2004.
- [9] A. K. Dewdney. In the game called Core War hostile programs engage in a battle of bits, Computer Recreations. *Scientific American*, May 1984.
- [10] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proc. of the ACM Symposium on Operating Systems Principles*, 2003.
- [11] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of the 10th Annual Network and Distributed System Security Symposium*, 2003.
- [12] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 1987.
- [13] J. B. Grizzard. *Towards self-healing systems: re-establishing trust in compromised systems*. PhD thesis, Atlanta, GA, USA, 2006. Adviser-Henry L. Owen, III.
- [14] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.
- [15] D. Hollingworth and T. Redmond. Enhancing operating system resistance to information warfare. In *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, volume 2, 2000.
- [16] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proc. of the 14th ACM conference on Computer and communications security*, 2007.
- [17] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008.
- [18] L. Litty and D. Lie. Manitou: a layer-below approach to fighting malware. In *Proc. of the 1st workshop on Architectural and system support for improving software dependability*, 2006.
- [19] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *13th USENIX Security Symposium*, 2004.
- [20] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. of the 15th USENIX Security Symposium*, 2006.
- [21] N. L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *14th ACM conference on Computer and communications security*, 2007.
- [22] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In *Proc. of the 2007 ACM symposium on Applied computing*, 2007.
- [23] S. regenerative Systems (SRS) Program Phase II. Cognitive immunity and self-healing. In *BAA 06-35 Proposer Information Pamphlet*. Defense Advanced Research Projects Agency, 2006.
- [24] R. Rivest. The MD5 Message-Digest Algorithm. Technical Report Request For Comments 1321, Network Working Group, April 1992.
- [25] S. J. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*. Prentice Hall, 2nd edition, 2002.
- [26] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *21st ACM symposium on Operating systems principles*, 2007.
- [27] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *20th ACM symposium on Operating systems principles*, 2005.
- [28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4), 2001.
- [29] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with Strider Ghost-Buster. In *Proc. of the International Conference on Dependable Systems and Networks*, 2005.
- [30] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a vmm-based usage control framework for os kernel integrity protection. In *Proc. of the 12th ACM symposium on Access control models and technologies*, 2007.
- [31] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. In *10th ACM SIGOPS European Workshop*, 2002.